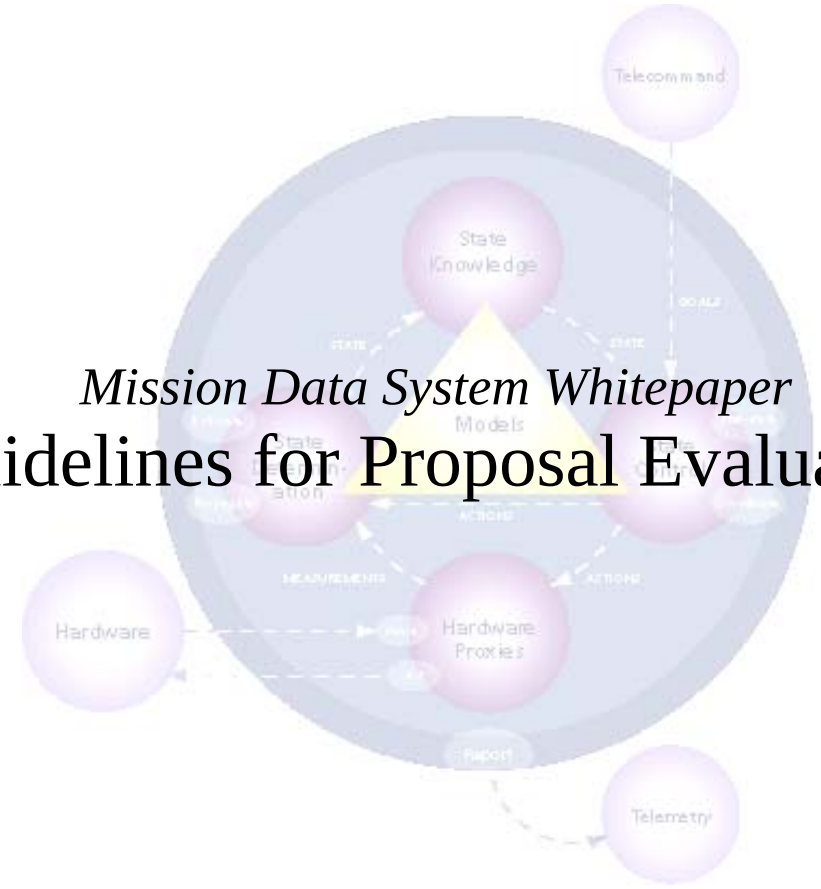


Mission Data System Whitepaper

Guidelines for Proposal Evaluation

Jet Propulsion Laboratory
California Institute of Technology

The Mission Data System (MDS) technology is designed to enable the development of highly reliable, software-intensive control systems. To achieve this goal, the MDS approach provides an engineering process and a unified set of capabilities that address both systems and software engineering concerns. The MDS approach differs from traditional engineering approaches that treat the systems and software disciplines as separate and distinct activities. And, MDS consolidates capabilities that are traditionally-engineered and planned as separate development investments. Consequently, *engineering managers may find it challenging to make an apples-to-apples comparison between an MDS proposal and a traditional proposal*.

This document poses a set of questions whose answers evoke the overall analytic, design and functional benefits that are integrated into MDS. Our goal is to help a technical manager make accurate cost and capability comparisons between proposals based on MDS and those based on traditional methods. These questions do not touch on every topic needed for a comprehensive analysis, but they suggest topics that should be covered in an objective comparison.

The following questions will be discussed:

- Does the proposed system have a regular architecture?
- Are the systems engineers and the software engineers speaking the same language?
- Is the software designed for reuse?
- What is the Ops Concept for the proposed system?
- How does the proposed system incorporate fault protection?

NOTE: The following discussion will refer to the "Top Software Issues" being addressed by the Software Engineering and System Assurance Office of the Office of the Under Secretary of Defense. A brief description of these issues appears in the Appendix.

Does the proposed system have a regular architecture?

Background

The MDS approach is anchored in a canonical architecture that gives a name and a place for the parts needed to build a system's applications. The architecture restricts the application designer to using specific types of parts and interconnections. These restrictions were derived from basic principles of system control and operation and have demonstrated recurring value over generations of system development.

Engineers who use MDS will construct a system rigorously according to a highly-regular, disciplined pattern. [1] The architecture guides the design from start to finish so that applications are approached architecturally and not as workarounds.

By comparison, a conventional approach typically defines architecture only at the subsystem level. Typically, a traditional architecture defines only *what* applications are needed and what low-level services will support their interconnections; it does not

[1] "Regular" refers to the use of a pattern that repeats according to rules.

provide guidance for *how* those applications should be built, or for establishing and maintaining their conceptual integrity. For example, a COMM engineer may maintain several representations of the spacecraft position and the Attitude Control engineer may maintain a entirely separate set of representations. As separate entities, these system elements may perform well, but when the design of a cross-cutting capability like fault protection begins, logical inconsistencies or oversights is likely to create unwelcome emergent behavior. and the system quickly becomes difficult to decipher. Without architectural guidance, the overall system may grow to become an unmanageable assembly of gadgets.

If architecture is based on a regular pattern, the *regularity* in design gives the engineering manager a powerful tool for estimating the cost and schedule for system development and integration. For example, during the State Analysis process, the engineering team decomposes the software into state-based elements derived from the system model. A typical state decomposition proceeds as follows: For each state-based element there is a state variable. For each state variable, there is an estimator. For each estimator, there are a set of measurement models and command models. And so on. The process continues according to the well-defined precepts of the methodology. The team explores the design space, eliciting the content and interfaces for each element, by simply repeating the process. The result is a detailed parts list that can be analyzed in terms of the number and complexity of parts needed.

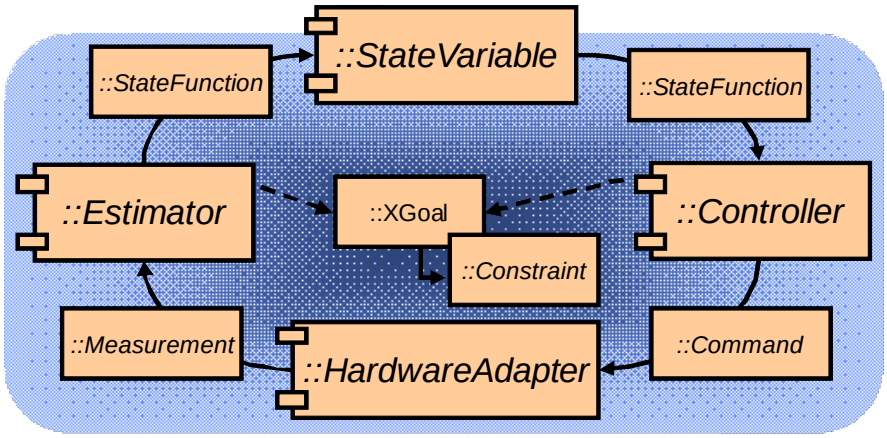


Figure 1: High-level view of MDS Canonical Architecture

By comparison, a conventional approach will generally provide cost estimates based on high-level functional descriptions and extrapolations from previous implementations of a similar application. The underlying basis of estimate is comparatively weak and tends to neglect low-level details that may prove to be significant. Historically, estimates based on past performance are unreliable—especially for novel applications.

Finally, regularity in design contributes significantly to maintenance and scalability. Changes to a regular system can be understood in terms of the parts that must be added or replaced. The architecture principles explain how new things should be added and provide a technique for understanding the impact of changes. And, since the MDS system model explains the relationship between the software, the underlying hardware and the environment, the MDS methodology provides a solid foundation for assessing system changes.

By comparison, conventional systems may not have design principles to guide change or assess the impact of a change. Without architectural guidance, changes may lead to a loss of integrity, where the resulting emergent properties may be difficult to predict or even understand. In practice, an *ad hoc* system often becomes brittle and very expensive to scale and maintain.

Suggested evaluation questions for comparisons

- Does the proposed system describe the architecture of its applications?
- Is there a regular organization of architectural elements and rules for how they are connected?
- Can application cost and schedule be explained in terms of the number and complexity of the parts needed?
- Can changes to an application be explained in terms of architectural modifications? How will changes in requirements be addressed by software?

Are the systems engineers and the software engineers speaking the same language?

Background

System engineers have overall responsibility for the design and development of the system. They define and analyze the system functional needs, decompose the system components, define component interfaces, provide the key functional algorithms and manage the operation and maintenance of the system. In short, they provide the requirement glue that ties all the other disciplines together.

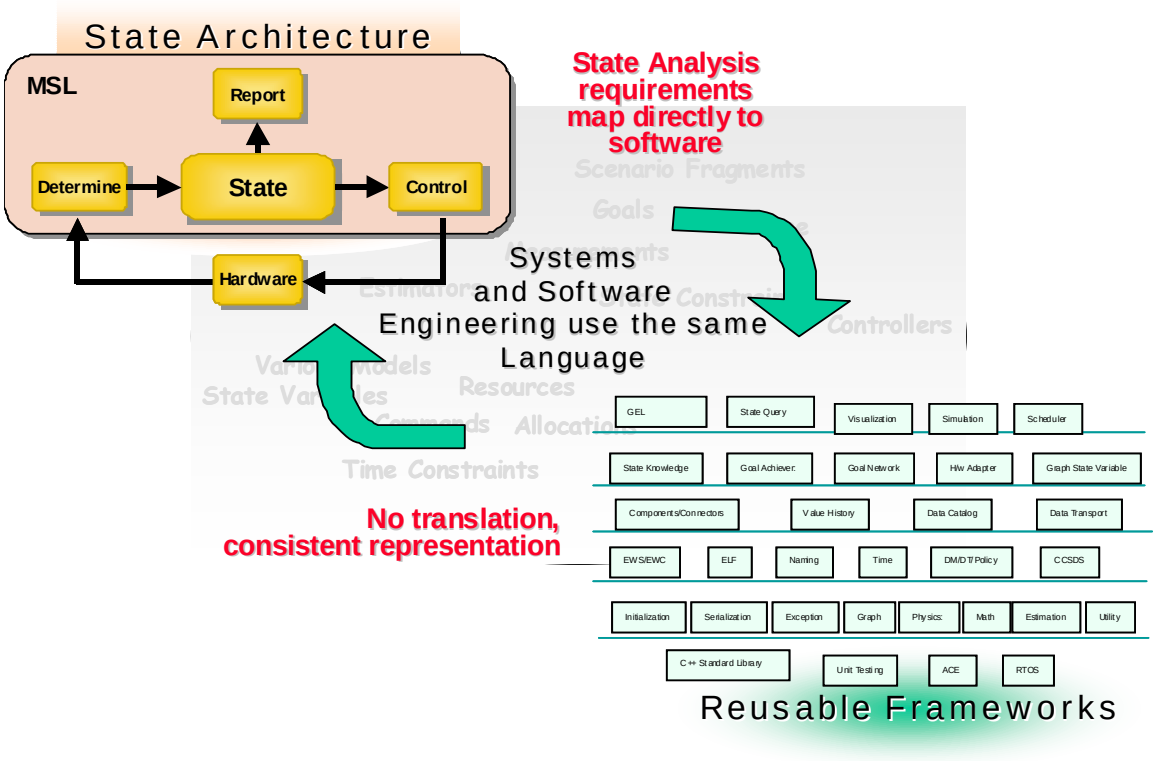
Software engineers use systems engineering specifications to design, implement and test the system's software. In theory, the software engineers will build the system specified by the system engineers.

However, conventional system engineering products do not translate directly into software. Software engineers must interpret functional requirement statements (typically "shall" statements) to build the actual system. This interpretation process is fundamentally error-prone, and mistakes of translation are exacerbated by inconsistencies and omissions in the requirements. In fact, software requirements are notoriously incomplete. The result is a big gap between systems and software engineering.

The MDS approach overcomes these traditional shortcomings by providing a common vocabulary to both systems and software engineers. When systems engineers conduct a MDS State Analysis, they capture system design and behavioral requirements in terms corresponding to the canonical architecture: states, commands, measurements, estimators, and controllers. Moreover, the methodology drives out the key design details and provides assurance that the analysis is complete and consistent.

These same architectural elements have been implemented in a software class library as the MDS Frameworks. When a programmer adapts the MDS Frameworks according a State Analysis, there will be one-to-one correspondence between the systems design and software design. The systems and software architecture and terminology are the same.

The system engineering effort maps directly to software, so interpretation is clear and no translation into an alternate structure is required.



Suggested evaluation questions for comparisons

- What is the relationship between the system architecture and software architecture?
- What guidance do software engineers receive for interpreting system requirements?
- What guidance does the project management have for estimating the impact of a requirement change on software?

Is the software designed for reuse?

Background

Software reuse refers to the practice of developing new applications with previously built software as a means of saving cost and schedule. Reuse may refer to architecture, requirements, tests, processes, and production rules as well as source code.

All software reuse is not equal. In an often-cited article, Poulin points out that software reuse can actually cost more than new development:

> "We have all seen experience reports that claim glowing results from reuse; however, reuse does not automatically provide the panacea that some reports seem to indicate."[2]

Why does reuse fail to deliver the promised benefit? The likely cause is that a project attempted to reuse software that was not designed for reuse. Often projects make reuse decisions by simply comparing the functions of an existing software package to the functions needed for the new application. Unfortunately for these projects, the evaluation was made without considering what needed to be modified, how it would be modified, or what the implication of those modifications might be.

However, not all software reuse is misguided. Some forms of reuse can be very effective. The devil lies in the modifications.

There is not a widely accepted classification of software reuse. However, for current purposes, we can distinguish among three types of reuse by focusing how the reused software would be modified. These three types are reuse of functional components, special purpose software platforms and adaptable frameworks.

- Reuse of functional components
  Components are elements that encapsulate some function. Some components are purely algorithmic and may be used without much modification. However, when a component depends on assumptions about the underlying platform, reuse may require extensive modification that breaks the ties to the original, unmodified code. The System Engineering Institute refers to this latter approach as "clone and own" because the project must then take full ownership of the code's maintenance and support.

  In general, fully-functional components will tend to be highly platform specific and will require modification if there are changes in the target system. An example of a platform-specific component might be an attitude control system or a power system. On the other hand, components that are designed for adaptation will tend to be usable on different target systems without modification. An example of an adaptable component might be a function in a math library, or a data compression algorithm.

- Reuse of special purpose software platforms
  A software platform is a foundation, sometimes called an architecture, on which applications are built. Example platforms might include operating system packaged with drivers for specialized devices, a set of services supporting a specific service-oriented architecture, or a virtual machine. A special-purpose platform typically has a strong dependency on a specific hardware package and is designed to isolate higher-level applications from the hardware. Another type of software platform may be highly optimized for performance or efficiency.

  If the platform is dependent on specific hardware, like special purpose sensors, reusing the platform with new hardware may require extensive modification. Or, in the case of a high performance or high efficiency system, applications may also

[2] See http://www.stsc.hill.af.mil/crosstalk/1997/07/reuse.asp

be strongly coupled to hardware, and reuse may require extensive modification. The software used for embedded mission-critical applications often fall into this category.

In recent years, mission projects have planned for considerable software savings based on software platform reuse assumptions. However, the inherited software proved to have strong dependencies on the underlying hardware, and, almost without exception, the customer project was required to "clone and own" the software. As a result, the software estimates proved overly optimistic and these projects were required to make extensive cost and schedule adjustments.

- Reuse of adaptable frameworks
  A software framework is a "reusable design for a software system (or subsystem). This is expressed as a set of abstract classes and the way their instances collaborate for a specific type of system."[3]

  Framework is an often-used, overloaded term applied to a host of reusable software packages. *While all frameworks are reusable, not all reusable software is a framework.*

  The key feature of a framework is that it consists of *frozen* spots and *hot* spots. The frozen spots define the overall architecture of a software system, that is to say its basic components and the relationships between them. These remain unchanged (i.e. frozen) for all uses of the application framework. On the other hand, hot spots (often called variation points) represent those parts where the programmers add the code to provide the required functionality.

  > "A significant benefit of using a framework is that it defines the places in the architecture where adaptations for specific functionality should be made... Having a good framework in place allows the developers to spend more time concentrating on the business-specific problem at hand rather than on the plumbing code behind it."[4]

  The MDS framework design goals closely match this description. The MDS Framework provides the basic capabilities needed to build a system on a variety of platforms. For example, the same code can be used in both the flight system and the ground system. The MDS Framework itself is *not* a completed capability, but a set of unfinished capabilities that are adapted and assembled into a complete application. MDS developers build on top of a stable library.

  > Note: this is distinctly different from the "clone and own" approach since the core assets remain stable during development.

Suggested evaluation questions for comparisons

- Are the reusable software assets designed specifically for the target application?

---

[3] See http://en.wikipedia.org/wiki/Software_framework

[4] See http://www.jot.fm/issues/issue_2004_09/article1/

- Does the software have strong dependencies on a specific hardware configuration?
- Has the modification methodology been specifically defined? Are the hot spots explicitly defined?
- Can the specific modifications be accomplished without disrupting the conceptual integrity of the system?
- What degree of modification is required for the proposed implementation? Changes only to comments? Internal functions? Interfaces? Is the code closely coupled to an underlying platform? (e.g. a device driver is, by definition, closely coupled.) Will parts of the code need to be restructured?
- How do you assess changes to the software if the platform changes?

What is the Ops Concept for the proposed system?

Background

There are two fundamental approaches to spacecraft operations: open loop and closed loop.

In an open-loop system, commands are sent to the spacecraft as sequence of timed actions that the spacecraft executes mechanically. In order to avoid unintended results, the uplinked sequence must be right: actuations must be correct, resources must be allocated, and flight rules must be maintained.

Open-looped systems are typically built with relatively simple, time-based sequencers. For example, a trajectory change maneuver sequence would consist of a timed series of commands that configure the propulsion valves, turn on the heaters, turn the spacecraft to the maneuver attitude, fire engines for the appropriate duration, and restore the system to its pre-maneuver state. Success depends on accurate ground predictions of spacecraft performance or on bounding conditions that guard against worst case behavior. If the sequence fails, it is typically abandoned and must be replanned to account for changed circumstances. In order to avoid failure, expensive ground simulations are developed to provide a higher reliability and improve the odds of success. Nonetheless, open-loop systems remain vulnerable to worrisome command errors, and considerable human rigor is needed to avoid issuing commands that endanger the mission. This slim margin of error places a heavy burden on the operational team. Historically, open loop works well during nominal operations for highly predicable missions with ample operational budgets.

In a closed-loop system, commands are sent to the spacecraft as statements of operator intent, or goals. A goal specifies *what* to do but not *how* to do it leaving commanding options open to the control system. The spacecraft uses onboard models to calculate system state from its sensors and then determines what commands are needed to put the spacecraft in the desired state. For example, a trajectory change maneuver goal consists of the desired state of the trajectory vector and the time the change to that state is desired. Because closed-loop systems are self-checking, success relies less on accurate or bounding predictions.

Since closed-loop systems are designed to operate properly without operator intervention, they work well for missions that operate in unpredictable environments and need on-board decision making. They are also desirable for missions with constrained operational budgets.

Closed-loop commanding is not a bolt-on capability. The control system must be able to reason about expressions of operator intent, and that must be integral to the architecture from the outset. Here are a few of the architectural qualities that facilitate reasoning about operator intent:

- Separation of intent from state knowledge and state knowledge from state control.
- Uniquely maintained knowledge of each state in the system
- Knowledge of state timeliness and uncertainty
- Conflict avoidance mechanisms

Without these, or comparable, design qualities, a flight system will not be well-suited for closed-loop operations.

> NOTE: In practice, most systems have some goal-based behavior, but usually only in local behaviors. Systems-level operations remains largely open loop. For systems, like MDS, *all* operator intent is expressed in goals. Consequently, MDS is often referred to as a goal-based system.

Fully closed-loop, or goal-based, systems provide a number of operational advantages over the time-based sequences of open-loop systems. Here are a few:

- Goal-based systems are self-checking and less expensive to validate and operate.
- Goal-based systems are safer because they have a built-in notion of what is safe and what is not safe.
- Goal-based systems reduce operational complexity and enable iterative operational workflow with the following benefits: fewer steps in the workflow, intent is preserved throughout the workflow so sequence revision does not require a plan restart, and plan changes can be made any time prior to uplink.
- Goal-based systems are flexible. They can be either time or event driven. They are designed to accommodate reordering of activities to improve resource usage.
- Goal-based systems are inherently more robust. Half-finished activities can be interrupted and resumed.
- Goal-based systems are built with more explicit applications of models. This simplifies system diagnostics. Operators have a more precise understanding of spacecraft state.
- Goals express intent specifications that can be checked formally for correctness and can reduce the likelihood of command errors.

MDS was engineered specifically to support goal-based as well as open-loop operations. Goals are a first-order concept in the architecture, so the operational concept is completely integrated with the canonical architecture, the State Analysis process, and the software frameworks. In other words, operations is not a bolt-on concern; rather, the engineering team will be analyzing and designing the operational capabilities starting at the earliest stages of the project.

Suggested evaluation questions for comparisons

- Does the proposed system rely, to any extent, on a open-loop Ops concept?
- Is that spacecraft command execution mechanism self-checking?
- Is there a mechanism for avoiding command errors?
- How labor intensive is the Ops concept?
- Can interrupted activities be resumed without ground replanning?
- Does the Ops concept and the flight system architecture support scalable autonomy? In other words, can the amount of autonomy in the Ops concept be varied?

How does the proposed system incorporate fault protection?

Background

A critical system must keep working. It must work in the presence of both internal errors, faults, and external hazards.

In a conventionally designed system, fault protection is composed of a set of monitors and a set of response mechanisms. The fault monitors watch for system and environmental symptoms of anomalous behavior. If a monitor detects a problem, a fault response mechanism halts the current activities and triggers a corrective action, like a swap to a backup device. Additional response mechanisms might then step in to restore the system to a safe state and perhaps attempt restoration of critical activities. Sometimes these corrective actions use the standard sequencing system, but often a special fault response mechanism kicks in.

On the surface, conventional fault protection appears to be quite straightforward. But in practice, the use of monitors and response mechanisms leads to the creation of the most complicated component in an otherwise complicated critical system. In particular, the behavior of a conventional fault protection system is often difficult to understand and nearly impossible to test.

According to Rasmussen, the problem stems from the underlying architectural assumption that fault protection is an appendage to the system. [5] To explain why, Rasmussen lists over two dozen issues that complicate the implementation of a monitor-response mechanism. Here are just a few:

- Many scattered symptoms may appear concurrently from one root cause, often masking the real culprit. There may be several possible explanations for observed difficulties, each with different ramifications or requiring competing responses.

[5] See https://pub-lib.jpl.nasa.gov/docushare/dsweb/Get/Document-316/08-031+GN%26C+Fault+Protection+Fundamentals.pdf

- Some faults may be hard to distinguish from normal operation, yet still cause long-term trouble.
- False alarms may provoke responses that are as disruptive as a genuine fault, so there is pressure to compromise on safety.
- Quirks or omissions in underlying functionality may conspire against the accomplishment of otherwise reasonable actions.
- Faults may create an urgent hazard to safety or critical operations that must also be handled at the same time.

Why the complications? According to Rasmussen, conventional architectures lack the features needed to deal with disrupted activities, or to reason about how the system with concurrent command activities will behave. In fact, the best that can be done with a conventional architecture is to attempt to test all paths—an approach that is intrinsically unachievable and astronomically expensive to attempt. Most projects address testing in a best-effort, piecemeal fashion. As a side effect of the testing issues, the numbers of monitors and responses are minimized which, in turn, means that much system state knowledge is outside the fault protection system. And, the issues proliferate.

There is a better way.

If the fault-protection system is an integral element of the architecture, the fault protection system can be viewed as part of the control system where faults are merely larger than usual deviations from control objectives. Since the design and operation of a control system is well understood, the fault protection system can be engineered in terms of states and objectives. This allows the fault-protection effort to benefit from the extensive body of practice in control and to capitalize on the control system architecture, methodology, and software framework. The fault protection would remain a challenging task, but integration into the architecture significantly reduces the *unknown-unknowns*.

MDS accomplishes the integration of the fault protection by using the same capabilities needed for goal-based commanding. This is possible because fault protection can be addressed simply as goals on state. And, goal failure, whether caused by a fault or normal events, is intrinsic to the architecture. The architecture further facilities the fault protection design because the system engineering method explicitly specifies the modeling basis of estimation and control for each state in the control system. In addition, architecture provides guidance for modeling consistency and support for root cause diagnosis. And last, but not least, the command execution mechanism provides for smart response to diagnosed faults so that activities can be rescheduled, not just terminated. For spacecraft with human occupants, activity resumption may prove to be critically important.

Suggested evaluation questions for comparisons

- Is the fault protection system integral to the architecture?
- If a fault occurs during an activity, does the system resume the activity after the fault response is complete?
- How are fault protection objectives expressed? Can those objective be validated as set? Can the interaction among objectives be understood?

- What is the process for restoring the control systems understanding of system state?

Summary

This document is intended to provide a technical management team with an evaluation tool for comparing an MDS-based system to a conventional system. Our goal is to depict the overall analytic, design and functional benefits that are integrated into an MDS approach and that may not be part of a competing approach.

In summary, the MDS key capabilities that will differentiate it from a traditional approach are:

- An architecture that explains how applications will be developed and maintained.
- A common language for systems and software engineering that eliminates a significant category of error.
- A software framework designed from the bottom up for reuse thus making overly optimistic reuse estimates less likely.
- A goal-based operational mechanism that reduces operation cost and provides greater dependability.
- An integral fault-protection system that provides "smart" fault response and helps engineers and operators reason about system faults.

All these benefits derive from the underlying architecture that provides a unique means of engineering the system as a *whole*, not as isolated subsystem parts.

Acknowledgement

The research described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

Appendix: DoD Software Issues and Gap Identification

In 2006, the Office of the Under Secretary of Defense (OUSD) established the *Software Engineering and System Assurance Office* inside the *Acquisition and Technology* Directorate to address the software issues that are plaguing the development of new DoD systems.

The principal motive for the creation of the Software Engineering and System Assurance Office was to address the growing role software was playing the development and maintenance of new systems. In particular, there are concerns about the growing cost of development and ownership associated with large software systems.

Historically the DoD has depended on the private sector to provide solutions that would reduce cost and improve quality. However, the marketplace has focused its investment on the profitable E-Business sector. Unfortunately, the E-Business tools provide little benefit to developers of complex, software intensive control systems. Recognizing a long-term trend, OUSD decided it must take a more active role in guiding the development of tools and technologies that are needed for the development of critical systems.

In recent presentations, Kristen Baldwin, the Deputy Director, Software Engineering and System Assurance Office has cited the top seven issues confronting developers of DoD Software Systems:[6]

1. The impact of requirements upon software is not consistently quantified and managed in development or sustainment.
2. Fundamental system engineering decisions are made without full participation of software engineering.
3. Software life-cycle planning and management by acquirers and suppliers is ineffective.
4. The quantity and quality of software engineering expertise is insufficient to meet the demands of government and the defense industry.
5. Traditional software verification techniques are costly and ineffective for dealing with the scale and complexity of modern systems.
6. There is a failure to assure correct, predictable, safe, secure execution of complex software in distributed environments.
7. Inadequate attention is given to total lifecycle issues for COTS/NDI impacts on lifecycle cost and risk.

Baldwin goes on to report on systemic issues that are significant contributors to poor program execution[7]

---

[6] From a report presented to National Defense Industrial Association. See http://www.ndia.org/Content/ContentGroups/Divisions1/Systems_Engineering/Draper_10_19.pdf

- Software requirements not well defined, traceable, testable
- Immature architectures, COTS integration, interoperability, obsolescence (electronics/hardware refresh)
- Software development processes not institutionalized, planning documents missing or incomplete, reuse strategies inconsistent
- Software test/evaluation lacking rigor and breadth
- Schedule realism (compressed, overlapping)
- Lessons learned not incorporated into successive builds
- Software risks/metrics not well defined, managed

[7] See http://www.ndia.org/Content/ContentGroups/Divisions1/Systems_Engineering/Baldwin_SoftwareAssurance.pdf